Creative Education, 2015, 6, 1456-1465
Published Online August 2015 in SciRes. http://www.scirp.org/journal/ce
http://dx.doi.org/10.4236/ce.2015.613146

Computers and Language Learning

Junia Rocha[1], Alexsandro Soares[2], Mauro Honorato[3], Luciano Lima[4], Nayara Costa[4], Elvio Moreira[5], Eduardo Costa[4]

[1]Department of Informatics, Federal Institute of Triangulo Mineiro, Patos de Minas, Brazil
[2]Department of Computer Science, Federal University of Uberlandia, Uberlandia, Brazil
[3]Department of Informatics, Federal Institute of Sao Paulo, Barretos, Brazil
[4]Department of Electrical Engineering, Federal University of Uberlandia, Uberlandia, Brazil
[5]Department of Education, Federal University of Uberlandia, Uberlandia, Brazil
Email: juniamagalhaes@iftm.edu.br, prof.asoares@gmail.com, maurojh@gmail.com, lucianovieiralimaster@gmail.com, asc.nayara@gmail.com, elvio.esm@hotmail.com, edu500ac@gmail.com

Received 30 May 2015; accepted 2 August 2015; published 5 August 2015

Abstract

This paper investigates how computers together with Internet technologies help people in the learning of languages. To achieve this goal, it analyses open source libraries that a teacher can use to build educational applications. The text contains a short discussion on how to build such tools, using methods of programming proposed by Richard Stallman and Paul Graham. It also shows that computers help to improve language skills in those children with low reading abilities. Finally, it provides an overview of linguistic and computational tools that a teacher can use to check a student's grammar. Of course, in order to build a practical grammar checker, the reader must have a working knowledge of Lisp and Prolog. In few words, the reader will not only see the magic of programs that understand English grammar, but learn how one can reproduce it.

Keywords

Rasch Model, Natural Language Processing, Automatic Grammar Checker

1. Introduction

When one hears about computer aided language learning, she thinks immediately in Artificial Intelligence, and machines that can talk, translate all English Wikipedia entries to Esperanto, perform automatic caption, and accept speech-to-text input. Everybody knows that such intelligent applications exist, and help millions of people in their dealings with a multi-language world. Therefore a large portion of the present paper concerns Artificial Intelligence and programming. However, the greatest help that computers bring to language learning is the possibility of publishing books.

How to cite this paper: Rocha, J., Soares, A., Honorato, M., Lima, L., Costa, N., Moreira, E., & Costa, E. (2015). Computers and Language Learning. *Creative Education, 6,* 1456-1465. http://dx.doi.org/10.4236/ce.2015.613146

While doing graduate studies in Cornell in the seventies, one of the authors of this work dedicated himself to space science and linguistics. His main concern at the time was learning Ancient Greek. As a space scientist, he had access to good computers to perform numerical calculations. However, electronic publishing did not exist at the time, and the powerful Cray computer available to engineers was of little use for reproducing Greek documents.

In 1984, Donald Knuth made TeX available (Knuth, 1984). TeX allowed for the typeseting of Greek books. Therefore, students could learn to read Greek by typesetting their own edition of Plato's Republic.

With the invention of eInk, publishing a book became even easier. Now students can carry around whole libraries in a device weighting less than 200 g. While reading a book, the student can touch a word whose meaning is unknown, and get its definition in 20 languages.

Another way that computers help language learning is to connect learners to native speakers through the Internet. For instance, the authors observe that children that are learning Chinese in the United States spontaneously contact people living in China.

Computer aided language learning has two kinds of tool. Typesetting, dictionary construction and connectivity do not require Artificial Intelligence. On the other hand, automatic caption, translation and speech-to-text input require a good deal of artificial intelligence.

Let us end this section with a short discussion on programming. Students of computer science learn their trade by trying to write programs from scratch. In summary, the students open a blank buffer in a text editor like emacs and start filling it with code. Stallman (2015) thinks that such an approach is wrong. In his opinion, the student should start with an open software application, and adapt it to her needs, fix bugs and extend its functionality. To make a long story short, if the computer scientist does not start with the accumulated work of linguists and teachers, she is doomed to failure.

There are many open source tools that a language teacher can use to build parsers. The easiest of these to install and use is the open source version of RASP (Briscoe, 2015). In order to make this conversation easier, let us introduce Nia, a fictitious female character that performs natural language processing for a living. Nia downloads and expands the archive in her work space, and runs the Makefile script in the application home directory.

```
~/wrk/rasp3os$ make
```

Since Nia has a limited knowledge of shell commands, she asks for help from Bob and Alice, who work in the Cryptography Department, as anyone familiar with the world of Computer Science fictitious characters already knows. After a few minutes, there appears the Clozure Common Lisp together with the RASP system in Nia's machine. Nia checks what comes ready to go.

```
~/rasp3os/scripts$ echo "cats chase rats" | ./rasp.sh -p'-os'
(|cat+s:1_NN2| |chase:2_VV0| |rat+s:3_NN2|) 1 ; (-8.936)
sparkle: 1
("S" ("NP" "cat+s:1") ("VP" "chase:2" ("NP" "rat+s:3")))
```

It seems that, after learning Lisp, Nia will be able to build a simple grammar checker on top of RASP.

When dealing with languages, one needs a parser. Following Stallman philosophy, the parser should be open source and written in a language that facilitates contributions from the linguist. The computer languages that satisfy this last requisite are Lisp and Prolog.

The focal point of this introductory section is that one should not start a programming project on an empty buffer. The most effective approach to programming is to build code on top of existing tools, as recommended by Richard Stallman.

2. Lisp

A small team that wants to write an educational natural language application should rely on one of the libraries available on line. However, it is a good idea to build a small prototype, in order to learn programming, and understand how libraries work.

Lisp has two features that one cannot find in other languages, and makes it specially suitable for Artificial Intelligence. For one thing, Lisp has a strong mathematical foundation. Mathematics does not become obsolete.

People use books written by Gauss 200 years ago, or by Archimedes, 2000 years ago, and they are considered state of the art. Although Lisp is the oldest computer language that is widely used, the very best works in Artificial Intelligence are coded in Lisp even today. After all, Lisp has remained unchanged for decades, while generations of mathematicians and physicists add code to complex applications.

The syntax of Lisp is extremely simple: Programs and data are represented by a list of elements between parentheses. When that list represents a program snippet, the first element is an operation, i.e., a command or a function call. When a list is data, it is prefixed by a quotation mark. Let us define a small vocabulary, and a few syntax rules.

```
;; Store this program in file cyk.lisp

(defparameterwrds
            '((cat n)
              (rat n)
              (thedet)
              (to prep) (chasestverb)))

(defparameter rules
            '( (pp prep np)
               (npdet n)
               (svtverbnp)
               (phrasenpsv)))
```

The parameter *wrds* represents a list of words, where each word is associated with its grammatical class. For example, *cat* is a noun, therefore its dictionary entry is the sublist (*cat n*). The grammar is written in the so called Chomsky Normal Form. The rule (*npdet n*) means that a noun phrase is made of a determiner followed by a noun.

Representing the dictionary and the grammar as lists means that the computer would need to loop through all entries. Loops are both inefficient and hard to code. They should be used only when necessary. Dictionary lookup must be performed with a hash table.

```
;; Continuation of the cyk.lisp file

(defparameter gr (make-hash-table :test 'equal))

(defparametervy (make-hash-table :test 'equal))

(loop for (hd nt1 nt2) in rules do(setf (gethash (list nt1 nt2) gr) hd))

(loop for (w cat) in wrds do (setf (gethash w vy) cat))
```

The above snippet creates two hashtables, that use the lisp predicate *#'equal* for comparison. This predicate is able to check whether two lists are equal.

To test the program, the linguist needs a text editor and a lisp. The text editor must be emacs. One can find emacs and the sbcl Lisp on the Internet.

Since Nia has already installed the RASP library, she decided to use it. A Read Eval Print Loop prompt appears on the window. Below one can see how Nia tested the program.

```
~/wrk/rasp3os$ bin/x86_64_darwin/ccl
Welcome to Clozure Common Lisp Version 1.8-r15286M
? (load "cyk.lisp")
#P"/Users/ufu/wrk/rasp3os/cyk.lisp"
? (gethash 'cat vy)
N
T
? (gethash '(det n) gr)
NP
```

It seems that everything is working fine. Now Nia will write a snippet that retrieves grammar rules, and use them to rewrite an input list. Of course, with such a small vocabulary in the hash table, the input sentence must be something like *the cat chases the rat.*

```
(defunrp(s) (gethash (list (first s) (second s)) gr ))

(defunky(input)
          (loop for s = input then (if (rp s) (cddr s) (cdr s))
            for hd = (if (rp s) (rp s) (car s)) while s collect hd))

(defunnxt(fn s &optional (z (funcallfn (car s))))
           (if (equal (car s) z) s (nxtfn (cons z s))) )

(defunnr (n) (lambda(guess)
        (- guess (/ (- (expt guess 2) n)(* guess 2.0)))  ))

(defun tree(words)
          (nxt #'ky (list(loop for w in words collect (gethash w vy) ) words)))
```

Lisp has an extremely simple syntax: The first element of a list is the operation and the other elements are arguments. For instance, (*rp s*) picks the left hand side of a rule, (*first s*) produces the first element of s, and so on.

The *nxt* program repeatedly applies a function to the first element of a series until the series converges to a fixed value. Before proceeding to syntactical analysis, let us test the *nxt* program with a subject that is easier than natural language, something like mathematics.

The *nr* is the famous Newton algorithm that calculates the square root of a number. Before trying to understand it, Nia checks whether it works.

```
* (load "cyk.lisp")
T

* (nxt (nr 16) #'equal '(3))
(4.0 4.0000014 4.003333 4.1666665 3)

* (car '(a x b e))
a

* (cdr '(a x b e))
(x b e)

* (cons 'top '(a x b e))
(top a x b e)
```

It seems that it works. The series converges to the square root of 16. Function *nxt* finds the limit of a series by repeatedly applying a function that adds the next element of a series to the head of list *s*. Of course, Nia will not learn Lisp in a short paper. However, she can understand an amazing fact. One can represent almost everything with lists. Nia has seen that she can use lists to represent grammar, sentences, arithmetic series and syntactical trees. Besides this, one needs only four functions to process lists.

The *car* and the *cdr*, in the theory of algebraic data types, are called selectors. These two functions take a list apart: (*car s*) calculates the head of *s*, while (*cdr s*) calculates the tail, which is the remaining part of the list when its first element is removed. The (*cons a s*) is called constructor, and builds a list whose car is *a* and the cdr is *s*. The predicate (*null s*) checks whether a list is empty.

Function *nxt* keeps adding elements to the head of the list until an element converges to the same value as the previous one. Function *fn* calculates the next element of the series. Nia noticed that *fn* is passed as a parameter to

nxt. This is necessary because this *fn* changes from one application to another.

Expression (*nr n*) builds a function that calculates the next approximation of the square root of *n*. Yes, Lisp functions can build other functions as easily as Python builds intergers. Lisp has two mechanisms that allow programs to build programs: closures and macros. There are two books that one can use to learn more about closures and macros. The first one was written by Paul Graham (Graham, 1993). Nia prefers Paul Graham's book, but the most popular one is Practical Common Lisp, by Peter Seibel (Seibel, 2005).

Let us go back to English grammar. One can use *nxt* to build a series of syntactic trees that converge to the representation of a phrase.

```
~/wrk/rasp3os$ rlwrap bin/x86_64_darwin/ccl
Welcome to Clozure Common Lisp Version 1.8-r15286M
? (load "infix.cl")
#P"/Users/ufu/wrk/rasp3os/infix.cl"
? (load "cyk.lisp")
#P"/Users/ufu/wrk/rasp3os/cyk.lisp"

? (tree '(the cat chases the rat))
((PHRASE) (NP SV) (NP TVERB NP)
        (DET    N      TVERB DET N)
        (THE CAT CHASES THE RAT))
```

The function *ky* picks each pair of symbols and checks whether there is a grammar rule able to rewrite the pair. After reading the first and second chapter of Practical Common Lisp, the interested reader will have no problem in understanding this short program.

3. Recovering from Blind Alleys

The problem with the parser described in program *cyk. lisp* is that the choice of a grammar rule may lead the algorithm down a blind alley, where there is no way to backtrack from the mistake. What is worse, the deterministic cyk algorithm has no mechanism to choose a rule with high probability of success. In fact, it does not even deal with probability.

Most people implements the cyk algorithm with arrays. This paper shows a list based implementation, because one can easily add backtrack to stateless data structures such as lists. The interested reader can use the screamer library to build a backtracking version of the *cyk. lisp* parser.

The work that gave rise to this paper uses both probabilities and backtrack. The rule with greatest probability of success is chosen first. This point deserves a comment. Consider the following rule:

$$S \rightarrow NP\ VP$$

The probability of *S* is the product of the probability of the rule by the probabilities of the subtrees *NP* and *SP*. To overcome the problem of estimating the probabilities of *NP* and *VP* before the full expansion of the tree, one solution is to accept backtracking in case of failure. With backtracking, a rough estimation of the subtree probabilities is acceptable.

4. Assessment

The previous section states that the probability of a construction occurring is used to resolve ambiguities in the grammar formalism. The main contribution of the present paper is a method for calculating the probability of a student committing a given mistake. To unify the framework, one describes mistakes as grammar rules. For instance, there are grammar rules for the lack of agreement between the verb and its subject. Therefore, a set of rules accept a sentence like *The cat chase mouses*, and tags it as a mistake.

In order to estimate probabilities, the teacher needs to model the student, and assess his/her evolution. The method explained here is amply used to evaluate learning.

In any kind of measurement, there is a variable that one wants to evaluate. Variables like weight, temperature or height can be measured directly with scales, thermometers, measuring tapes, calipters, etc. Unobservable va-

riables like skill and difficulty are not so easy to measure. One can describe such latent variables, but cannot compare them to a standard meter, since they lack physical dimensions. However, one needs to assess them to appraise student evolution.

In order to estimate the value of a latent variable, Rasch, Lord and Lazarsfeld developed independently a branch of statistics known as Measurement Theory. There is strong evidence found in Measurement Theory that leads educators to consider its superiority over classical test theory. Therefore, making it the preferred method for scoring high stake tests, like SAT.

4.1. Scales

Let us consider two students *C* and *V*. Suppose a teacher wants to discover the most common types of errors her students commit. In order to do this, one needs to compare the ability of the student in relation to a given grammatical rule.

The teacher's best option would be to write a grammar for mistakes. Let us examine exactly what a grammar for mistakes is. A very commoner or among students of English as a Second Language is lack of agreement between subject and verb. For instance, the subject may be in the third person singular and the verb in the plural: *She walk in beauty*. One can write a grammar that accepts this kind of sentence, and use it to compare how often it appears in texts that two students *C* and *V* have written. Let us ignore those results where both *C* and *V* commit the same mistake or both of them avoid it.

At first glance, it may seem strange that the counter ignores when an instance of the error occurs with both students. To understand that such a procedure does not alter the distance between the two students, let us consider **Table 1**.

C committed this particular kind of mistake 5 times. *V* made the mistake 8 times. The difference is 3. If one ignores the cases where both *C* and *V* hit the mistake together, then the final count for *C* drops to 2, and for *V* is reduced to 5. The difference between them is still 3.

The probability of *C* committing a mistake *r* is given by P_{cr} and the probability of avoiding it can be calculated by $(1-P_{cr})$. On the other hand, the probabilities of *V* committing and avoiding the mistake are given by P_{vr} and $(1-P_{vr})$ respectively. Let N_{10} be the notation of how many times *C* stumbles upon a mistake and *V* avoids it. On the same token, let N_{01} denote the number of times that *C* makes a mistake and *V* stays away from it. The ratio between N_{10} and N_{01} is given below

$$\frac{N_{10}}{N_{01}} = \frac{P_{cr} \times (1-P_{vr})}{(1-P_{cr}) \times P_{vr}}$$

4.2. Specific Objectivity

One can say that *C* is more prone to mistakes than *V* if the rate N_{10}/N_{01} does not change with the kind of mistake. This property is called specific objectivity and when it holds, one has the equality below.

$$\frac{P_{cr} \times (1-P_{vr})}{(1-P_{cr}) \times P_{vr}} = \frac{P_{cs} \times (1-P_{vs})}{(1-P_{cs}) \times P_{vs}}$$

Odds is the ratio of the probability of an event occurring to the probability of it not occurring. One can rewrite this equality in order to obtain the odds of *C* committing the mistake *r*

$$\frac{P_{cr}}{(1-P_{cr})} = \frac{P_{cs}}{(1-P_{cs})} \times \frac{(1-P_{vs})}{P_{vs}} \times \frac{P_{vr}}{(1-P_{vr})}$$

4.3. Origin

The next step is to choose the origin for the measurement scales that one intends to introduce. Let us consider a

Table 1. VS agreement, where 1 represents a mistake.

C	1	0	0	1	0	0	1	1	1	0
V	1	1	1	0	1	1	0	1	1	1

student *o* whose tendency to make mistakes matches the easiness of anitem *o*. In this case, the student will stumble upon the mistake in half of the trials, and the error will fail to defeat the student for the other half. This student is said to be at the origin of the inability scale, and the item is at the origin of the easiness scale. Since the student commits the mistake half of the times, the probability of failure is $P_{oo} = 0.5$. Let us compare *C* with the student of the origin.

$$\frac{P_{cr}}{(1-P_{cr})} = \frac{P_{co}}{(1-P_{co})} \times \frac{(1-P_{oo})}{P_{oo}} \times \frac{P_{or}}{(1-P_{or})}$$

Since P_{oo} is 0.5 one has that $(1-P_{oo})/P_{oo} = 1$. Therefore

$$\frac{P_{cr}}{(1-P_{cr})} = \frac{P_{co}}{(1-P_{co})} \times \frac{P_{or}}{(1-P_{or})}$$

Let's take the logarithm from both sides of this equation

$$\ln\left(\frac{P_{cr}}{(1-P_{cr})}\right) = \ln\left(\frac{P_{co}}{(1-P_{co})} \times \frac{P_{or}}{(1-P_{or})}\right)$$

$$\ln\left(\frac{P_{cr}}{(1-P_{cr})}\right) = \ln\left(\frac{P_{co}}{(1-P_{co})}\right) + \ln\left(\frac{P_{or}}{(1-P_{or})}\right)$$

$$\ln\left(\frac{P_{cr}}{(1-P_{cr})}\right) = \ln\left(\frac{P_{co}}{(1-P_{co})}\right) - \ln\left(\frac{(1-P_{or})}{P_{or}}\right)$$

If one defines

$$F_c = \ln\left(\frac{P_{co}}{(1-P_{co})}\right)$$

$$E_r = \ln\left(\frac{(1-P_{or})}{P_{or}}\right)$$

The equation becomes:

$$\ln\left(\frac{P_{cr}}{(1-P_{cr})}\right) = F_c - E_r$$

Notice that F_c does not depend on the grammar rule *r* and E_r does not depend on the student *c*. This finding is the greatest contribution made by Georg Rasch to the Measurement Theory (Wright & Mok, 2004).

The definition of the logarithm yields the following expression for the odds of committing a given mistake:

$$\frac{P_{cr}}{(1-P_{cr})} = \mathrm{e}^{F_c - E_r}$$

$$P_{cr} = \mathrm{e}^{F_c - E_r} - P_{cr} \times \mathrm{e}^{F_c - E_r}$$

$$P_{cr} \times \left(1 + \mathrm{e}^{F_c - E_r}\right) = \mathrm{e}^{F_c - E_r}$$

$$P_{cr} = \frac{\mathrm{e}^{F_c - E_r}}{\left(1 + \mathrm{e}^{F_c - E_r}\right)} \text{ Logistic equation}$$

One often refers to parameters F_c and E_r as an individual's inability and an item easiness respectively.

4.4. Calibration

In the previous section, the reader saw that probability depends on parameters like easiness and inability. Cali-

bration is thus the process of determining these parameters. To meet this goal, an iterative algorithm must force raw data onto the logistic curve.

The first step of the iteration calculates row and column averages to estimate initial easiness and inability vectors for the data matrix *Xn*.

```
(defparameterXn
          #2A (1.0 1.0 0.0 1.0 0.0)
         (0.0 0.0 1.0 1.0 0.0)
         (0.0 1.0 1.0 1.0 1.0)
         (0.0 0.0 1.0 0.0 0.0)
         (0.0 0.0 0.0 1.0 0.0)
         (0.0 0.0 1.0 1.0 1.0) ))

(defparameter d0 (array-dimension Xn 0))

(defparameter d1 (array-dimension Xn 1))

(defun hits(m)
           (make-array (list d0) :initial-contents
             (loop for i from 0 below d0 collect
               (loop for j from 0 below d1
          when #i(m[i,j]==1) sum 1.0 into s1
                 finally (return #i(s1 /d1))) )))

(defunina_logit
           (vet&optional (v_logit (make-array d0)))
           (loop for i from 0 below d0 do
         #i(v_logit[i]=log(vet[i]/(1-vet[i])))) v_logit)


(defun misses(m)
          (make-array (list d1) :initial-contents
            (loop for j from 0 below d1 collect
              (loop for i from 0 below d0
          when #i(m[i,j]==1) sum 1.0 into s1
                 finally (return #i(s1 / d0))) )))

(defuneas_logit(vet &optional(v_logit(make-arrayd1)))
          (loop for i from 0 below d1 do
         #i(v_logit[i]=log((1-vet[i])/vet[i]))) v_logit)
```

The next step is to adjust the easiness vector by subtracting the average from each element. The function *probability* calculates the odds, and stores its value in a two dimensional array.

```
(defunavg-vector (vet)
          (loop for x across vet
        sum x into s count x into c
            finally (return (/ s c))))

(defunadj_dif_logit (vet avg)
        (map 'vector (lambda(x) (- x avg)) vet))

(defun probability (A Dadj&optional (m (make-array (list d0 d1))))
           (loop for i from 0 below d0 do
             (loop for j from 0 below d1 do
         #i(m[i,j] := (exp (A[i] - Dadj[j])) /
                      (1.0 + (exp (A[i] - Dadj[j])))) )) m)
```

In order to update the inability and easiness vectors,one must calculate the residual between the current andthe previous probability matrix.

```
(defun residual(mi me &optional (rs (make-array (list d0 d1))))
          (loop for i from 0 below d0 do
            (loop for j from 0 below d1 do
        #i(rs[i,j] := mi[i,j] - me[i,j]) )) rs)

(defunresidual_sum(m &optional (sum (make-array d0)))
          (loop for i from 0 below d0 do
            (loop for j from 0 below d1 do
        #i(sum[i]:=sum[i]+m[i,j]))) sum)
```

Now, one needs to calculate the variance of the probability matrix.

```
(defun variance (m &optional (mv (make-array (list d0 d1))))
          (loop for i from 0 below d0 do
            (loop for j from 0 below d1 do
        #i(mv[i,j]:=m[i,j]*(1-m[i,j])))) mv)

(defunsum_row_mat(m)
          (make-array (list d0) :initial-contents
            (loop for i from 0 below d0 collect
              (loop for j from 0 below d1 sum
                #i(-m[i,j])))) )

(defunsum_col_mat(m)
          (make-array (list d1) :initial- contents
            (loop for j from 0 below d1 collect
              (loop for i from 0 below d0 sum
        #i(-m[i,j])))) )
```

After summing up the residual and variance for each of the two dimensional arrays along each row, one is ready to update the easiness and inability vectors. These steps must be repeated until the sum of the squares of the residuals becomes sufficiently small.

```
(defunnewF(A rsvrc
          &optional (nAbil (make-array d0)))
            (loop for i from 0 below d0 do
            #i(nF[i]:=A[i]-rs[i]/vrc[i])) nF)

(defunnewE (D rsvrc&optional (nDif (make-array d1)))
            (loop for i from 0 below d1 do
            #i(nE[i] := D[i]-rs[i]/vrc[i]) ) nE)
```

The calibration algorithm produces **Table 2** containing the probabilities of each student committing a given mistake.

5. Conclusion

The authors are convinced that natural language processing has reached a stage that makes building of automatic grammar checking possible. Another interesting application of these new technologies is the construction of models that facilitate the planning of grammar drilling. The abstract of the present paper was written by a

Table 2. Probabilities.

	You + is	She + go	It + stay	He + have	She + give	Inability
Porgy	0.2329	0.4890	0.8640	0.9525	0.4890	0.6810
Std 2	0.0784	0.2113	0.6401	0.8489	0.2113	−0.5463
Std 3	0.5267	0.7781	0.9588	0.9866	0.7781	1.9686
Std 4	0.0185	0.0561	0.2831	0.5550	0.0561	−2.0104
Std 5	0.0185	0.0561	0.2831	0.5550	0.0561	−2.0104
Std 6	0.2329	0.4890	0.8640	0.9525	0.4890	0.6810
Easiness	2.0564	0.8420	−1.2528	−2.4914	0.842	

person with limited knowledge of English, and corrected by natural language processing tools. The reader will find the original text below.

> This paper investigates how computers and communication can help people in learning languages. The authors will present both tools developed by themselves and by third parties. The text contains a short discussion on how to build such tools, using methods of programming proposed by Richard Stallman and Paul Graham. A longitudinal survey that the authors performed along three decades shows how computers improved the language learning environment. This article will provides enough information about linguistic and libraries for the reader building a program able to check of English prose composition. Of course, in order to build a practical grammar checker, the reader must have a working knowledge of Lisp and Prolog. In fewer words, this article intends not only to show the magic of programs that understand English grammar, but reveal how one can reproduce the effects.

References

Briscoe, T., Buttery, P., Carroll, J., Medlock, B., Watson, R. Andersen, O., & Parish, T. (2015). *RASP, a Robust Parsing System for English*. Cambridge: iLexIR. http://users.sussex.ac.uk/~johnca/rasp/

Graham, P. (1993). *On LISP: Advanced Techniques for Common LISP*. Upper Saddle River, NJ: Prentice Hall.

Knuth, D. E. (1984). *The TEXbook*. New York: Addison-Wesley Professional.

Seibel, P. (2005). *Practical Common Lisp*. New York: Apress. http://dx.doi.org/10.1007/978-1-4302-0017-8

Stallman, R. (2015). The Best Way to Learn Programming. https://www.youtube.com/watch?v=dvwkaHBrDyI

Wright, B. D., & Mok, M. M. C. (2004). *An Overview of the Family of Rasch Measurement Models in Introduction to Rasch Measurement* (pp. 1-24). Maple Grove: JAM Press.